Sentiment Analysis of Movie Reviews

Brianna Atayan
1632743
batayan@ucsc.edu

Colin Maher
1432169
csmaher@ucsc.edu

Lily Nguyen
1596857
lnguye78@ucsc.edu

CMPS 142
07 December 2018

1 Abstract

Sentiment analysis is the process of classifying the opinions, attitudes, or emotions expressed in a piece of text. In this project, we sought to build a machine learning model that could predict the sentiment of phrases from movie reviews.

The training dataset was provided for us, and was taken from the Rotten Tomatoes movie reviews corpus. The data was pre-labeled, with every phrase being classified numerically as 0 (negative), 1 (somewhat negative), 2 (neutral), 3 (somewhat positive), or 4 (positive).

We implemented our model in Python, using a mixture of our own code and outside libraries. We pre-processed the data, employed various feature extraction methods (such as TF-IDF and Bag of Words), and implemented an ensemble classifier that took the vote of the individual classifiers (a Naïve Bayes classifier, a Support-Vector Machine classifier, and a Logistic Regression classifier) to make a prediction about the sentiment of a given text phrase.

2 Tools Used

- Scikit Learn: Our group utilized Scitkit Learn's highly customizable classifiers, such as Multinomial Naive Bayes, Logistic Regression, and Support Vector Machines, which were inserted into Scikit's Voting Classifier as an ensemble method. We also used Scikit's TF-IDF and Count Vectorizers to extract notable features from our instances.
- Pandas: Our group needed the Pandas library to manipulate data tables in a .csv format. We mostly utilize Pandas' read and write functions to extract our test and training data, and then write the output to a new .csv file.
- NumPy and SciPy: These two libraries are dependencies of Scikit Learn, so they were necessary for Scikit's functions to execute properly.
- Natural Language Tool Kit (NLTK): We only used one tool, called WordNet Lemmatizer, to perform lemmatization on the text data to consolidate words with the same root lemma.

3 Data Pre-Processing

Data pre-processing is the first step in any machine learning problem. The raw data must be cleaned before it can be analyzed; if there is no standard format for the data, then the model will have a difficult time drawing conclusions from the data or will perhaps be completely unable to work with the data.

We began by implementing our own pre-processing methods. We started with tokenizing each text phrase into separate words, thereby preparing each instance for feature extraction. Then, we converted all words to lowercase, removed punctuation, and reduced each word to its lemma (using the NLTK lemmatizer). By performing these transformations on the text, we hoped to reduce the number of features and increase the strength of other features. For example, without cleaning the data, tokens like *good* and *Good* may be counted as separate features, or words like *terrible* and *terribly*. But by performing these pre-processing methods, we can condense what are essentially the same word into one token.

We chose not to use built-in tools to remove stop-words because some stop-words may change the sentiment of a phrase (.e.g. *movie was not good* vs *movie good*). We also hoped that with TF-IDF as one of our feature extraction methods, meaning-less words that have a high document frequency (like *the*, *a*, *an*) would naturally receive lower scores.

We also experimented with using the pre-processing methods built into Scikit's feature extractors (Count Vectorizer for Bag of Words and TFIDF Vectorizer for TF-IDF), which do not include lemmatization. We found mixed results; the best result was obtained by using our own pre-processing methods for TF-IDF but using the pre-defined pre-processing methods for Bag of Words, increasing our average cross-validation F1-score by 0.1 - 0.3%.

4 Feature Extraction

- Term Frequency - Inverse Document Frequency (TF-IDF) (Unigrams and Bigrams): After some research, we found that TF-IDF is a common choice of feature extraction for sentiment analysis. We wanted a feature extractor that would naturally filter out common words that do not affect the sentiment of a phrase, but without out-right excluding them (as in stop-words removal). We started with only having unigram features, but found our model's performance increased when we allowed for bigram features as well.

- Bag of Words (Unigrams and Bigrams, Trigrams (removed)): This is also a popular choice for feature extraction when dealing with sentiment analysis. When we added this as a second feature extraction method, on top of TF-IDF, we found that the performance of our model on the training set jumped about 6%. When we included bigram features, the performance increased nearly 10%. We experimented with including trigram features but, although it did increase the performance of our model on the training set, we ultimately left it out for fear of overfitting on the training data.

- Positive + Negative Word Count (removed): This feature extraction method counted the number of positive and negative words in each phrase by comparing each word to a list of words found on a NLP blog. This gave us only a marginal improvement in our accuracy, so we removed it in favor of other features.

- Linguistic Inquiry and Word Count (LIWC) (removed): We experimented with adding LIWC features to the vectors, using existing code and dictionary data we had from CMPS 143, the Natural Language Processing class. Unfortunately, these added features did not help our scores and actually decreased our accuracy by about 3% – and thus it was removed.

5 Machine Learning Model

The first approach we took was to take a set of three classifiers (Multinomial Naïve Bayes, Support-Vector Machines, and Logistic Regression) and utilize an ensemble method that would vote on the final label. These classifiers were built using the Scikit classifier libraries. We decided on these three as the individual classifiers because after some quick research, it seemed that these were three of the most popular models to use in sentiment analysis. We chose to create a voting ensemble classifier out of them because we didn't know how the individual models would perform, and without enough time to perfect the individual classifiers, we hoped the voting system would help balance the strengths of classifiers against their weaknesses.

After the performance results, we decided to give weights to the individual classifiers. We saw that the Naïve Bayes model was consistently underperforming compared to the others (about 10% to 20% worse in accuracy) on the training data, while SVM was consistently performing the best. Thus, we gave the most weight to SVM and the least weight to Naïve Bayes. However, after evaluating our model's performance with cross-validation, we realized that our model was performing too well on the training data and would probably not generalize well; testing directly on the training data, we were seeing an accuracy and F1-score of about 95%, but with three-fold cross-validation, we were seeing a result of about 61%. SVM and Logistic Regression had been performing exceptionally on the training data, but with three-fold cross-validation, we

found the performance of all three individual models was much worse and more similar, with Naïve Bayes performing slightly better than the rest – the opposite of what we had been seeing earlier.

Realizing that our model was possibly overfitting to the training data, we attempted to modify the parameter values for each of the classifiers to add stronger regularization. For Naïve Bayes, we tested our model using different values for the additive Laplace smoothing parameter; we found the best results between 1.0 and 1.5. However, the differences within that range were so minimal, we decided to leave the parameter at the default value of 1.0. For Logistic Regression, we attempted to increase the penalty value of the regularizer; we saw some improvement as we increased the penalty value (about 0.5% increase in the cross-validation score), and thus we changed the penalty value to 1.5. For SVM, we tried changing regularization method from L2 to L1. This also produced a slight improvement (also about 0.5% increase in the cross-validation score), so we made the change permanent.

Eventually, we decided to try implementing another individual classifier, Multi-Layer Perceptron, to compare its performance with the voting classifier's. Training the neural network took too long to run (over an hour), and we believed that any possible increase would not be worth the time to run it in the scope of our project.

After considering other methods, we finally decided to continue with our current model since it was reliable, quick, and had capacity to be modified to try to increase performance.

6 Final Prediction Set-up

As of testing time, our model uses an ensemble method comprised of Naïve Bayes, SVM, and Logistic Regression models, as well as a different variety of feature extraction methods. Our program starts by pre-processing the training and testing data using tokenizing methods that converts a phrase into a list of words, removes punctuation, non-alphabetic tokens, and lemmatizes the tokens, which is done by the WordNet Lemmatizer found in NLTK. We then use feature extraction methods on both data sets, such as TF-IDF and Count Vectorizer, which both use unigram and bigram methods to gather features. The training data is used to train our ensemble method of the three aforementioned models, where we use the default options for Multinomial Naive Bayes, changed the regularization method to L1 and set dual to false for Linear SVC, and set Logistic Regression to handle multi-class data sets and increased its regularization penalty. We then make predictions on the test set using the trained model. Our ensemble method assigns weights to the classifiers, with Logistic Regression set to 3, Linear SVC set to 5, and Naive Bayes set to 1. The ensemble method makes its predictions using a hard voting system.

7 Results

Model Performance on Entire Training Data

	Naïve Bayes	SVM	Logistic Regression	Voting Ensemble
Accuracy	**0.7288**	**0.9411**	**0.9324**	**0.9411**
Precision	**0.7401**	**0.9408**	**0.9324**	**0.9408**
Class 0	0.5566	0.9240	0.9248	0.9240
Class 1	0.6299	0.9296	0.9287	0.9296
Class 2	0.8454	0.9504	0.9343	0.9504
Class 3	0.6557	0.9342	0.9323	0.9342
Class 4	0.5937	0.9265	0.9323	0.9265
Recall	**0.7288**	**0.9411**	**0.9325**	**0.9411**
Class 0	0.6508	0.8685	0.8504	0.8685
Class 1	0.7132	0.9191	0.9006	0.9191
Class 2	0.7626	0.9733	0.9741	0.9733
Class 3	0.6926	0.9145	0.8969	0.9145
Class 4	0.6704	0.8760	0.8543	0.8660
F-1 Score	**0.7325**	**0.9407**	**0.9319**	**0.9407**
Class 0	0.6000	0.8954	0.8860	0.8954
Class 1	0.6689	0.9243	0.9145	0.9243
Class 2	0.8019	0.9618	0.9538	0.9618
Class 3	0.6736	0.9242	0.9143	0.9242
Class 4	0.6297	0.9005	0.8916	0.9005

Model Performance with Cross-Fold Validation (CV = 3)

	Naïve Bayes	SVM	Logistic Regression	Voting Ensemble
Average F-1 Score	0.5754	0.6278	0.6309	0.6277

Class Count

Class 0	Class 1	Class 2	Class 3	Class 4
4914	19050	55885	22960	6433

8 Conclusion

We found that it's best to not be too confident in a model until k-fold cross validation is used to check the model's ability to generalize. As we can see from the results, our model performs exceptionally well when trained and tested on the entire dataset – but performs much more poorly when trained and tested with k-fold cross validation. Interestingly, SVM and Logistic Regression perform significantly better than Naïve Bayes on the training data (about a 20% difference), but when it comes to cross-validation metrics, the difference between their performances is only about a 5 - 6% difference. As mentioned earlier, a likely cause for this discrepancy between the training performance and the cross-validation performance is overfitting. This suggests that the parameters for the models – particularly the ones dealing with regularization – should have been explored more thoroughly. All in all, we expect that our model will perform only okay on the actual test data.

9 Ideas for Future Work

The main problem we encountered in our project was the time it took to run our program when we wanted to print the performance metrics. At the worst, it took over an hour and had to be left running overnight. It was mainly the cross-validation that slowed our program, and this caused us to reduce the number of folds we used (from 5 to 3) and forced us to only print one performance metric each cross-validation test (F1-score). We attempted to find a better method or a way to optimize Scikit's cross-validator, but were unable to do so. In the future, we would like to explore other options – perhaps implement our own method

– so that we can either increase the speed of our program or at least obtain other additional performance metrics for each cross-validation test.

Some of the models that we wanted to use were not practical because of how long they took to run. We attempted to train neural networks as well RBF SVMS but given our limited computing power and our massive feature vectors these algorithms never terminated. So in the future we could use a cluster of servers or a supercomputer to train these models. We also thought of using word2vec to produce word embeddings, but were not able to get to implementing it.

We also would have liked to test our model with different ensemble methods. For example, instead of using a voting ensemble classifier, we could have tried a boosting method, such as AdaBoost, which may have helped with overfitting.

Also, looking at the class count of our training data, we can see that the data instances were not uniformly distributed. For classes 0 and 4 we had thousands of data instances, but for classes 1, 2, and 3 we had tens of thousands of instances. It is certainly possible that the imbalance in our dataset affected our training model, and we would have liked to have implemented some methods to alleviate the imbalance (such as sub-sampling or over-sampling).

10 Diversity: About the Authors

Our group has some obvious differences such as Lily and Brianna being female, and Colin being male. We are all of different ages despite all being seniors, due to being transfer students, and while we are all of Asian descent, Colin and Brianna are of mixed ethnicities, being Irish/Asian American and Filipino/Vietnamese respectively. However, our main differences as a group come from where we grew up and our family backgrounds. Lily's parents are first generation immigrants, and her mother has a master's degree. Colin's parents are both college educated, with his father also having a master's degree and being an immigrant from Ireland. Brianna's parents did not go to college, so she is a first generation college student, and her families on both sides are first or second generation immigrants. Lily grew up in southern California, while Brianna and Colin have both lived in the California Bay Area their whole lives.